Formal Verification of Security Properties of Smart Card Embedded Source Code

June Andronick[1], Boutheina Chetali[1], and Christine Paulin-Mohring[2]

[1] Axalto, Smart Cards Research
{jandronick,bchetali}@axalto.com
36-38, rue de la Princesse, BP45, 78431 Louveciennes Cedex, France
[2] Université Paris-Sud
paulin@lri.fr
Laboratoire de Recherche en Informatique, UMR 8623 CNRS
Bâtiment 490, F-91405 Orsay Cedex, France

Abstract. This paper reports on a method to handle the verification of various security properties of imperative source code embedded on smart cards. The idea is to combine two program verification approaches: the functional verification at the source code level and the verification of high level properties on a formal model built from the program and its specification. The method presented uses the Caduceus tool, built on top of the Why tool. Caduceus enables the verification of an annotated C program and provides a validation process that we used to generate a high level formal model of the C source code. This method is illustrated by an example extracted from the verification of a smart card embedded operating system.

Key words: Theorem Proving, Smart Card, Security, Source code verification, Formal Methods.

Introduction

In domains where security is a major issue, as in the smart card world, the need of confidence in the programs developed is increasing dramatically. This leads to a strong development of methodologies and tools which aim at strengthening this confidence. In particular, formal methods are proposed to provide formal verification of the correctness of crucial and sensitive programs.

Several approaches have been studied for the formal verification of systems. A first approach consists in building a model of the target system in a formal framework and in reasoning about the model in the same framework. The weakness of this approach lies in the confidence in that the model actually represents the system. However, some methods, like the automatic generation of source code, enable to strengthen this link between the model of the system and the code implementing it.

Another approach consists in verifying directly the source code of the system implementation. Functional properties of the system are defined by inserting

annotations in the code and a proof obligation generator is used to verify these properties.

The idea presented in this paper is to combine the two approaches, in order to prove *global security properties* on a *verified* model. In other words, the model consists in the code *specification*, that is the set of annotations, and the source code verification method is used to prove that the specification, i.e. the model, is verified by the implementation. This proof constitutes a *formal* link between the model and its implementation. Hence, the high level verification can be done on the *verified* model.

Our approach is somehow similar to the one proposed by the JCVM tool (see [7]) generating a formal model from a Java Card source code. The main difference, besides the fact that we are here interested in the C language, is that we use only the formal specification, whereas the JCVM tool build a model of the program itself, which may make the proofs heavier.

We use the Caduceus tool ([21,22]), built on top of the Why tool ([19,20]), which provides a *multi-prover* formal framework for the verification of C programs. Its architecture enables the definition of an *automatic* generation of a high level model in a formal environment (the proof assistant Coq [32]) where the verification of security properties is possible.

We use this method for the formal verification of an operating system module embedded on a smart card. Due to its central position in the architecture of smart cards, its validation is crucial for the confidence in the whole system. For intelligibility reasons, only a simplified case study will be presented in this paper, though a real embedded operating system module has been verified.

The paper is organised as follows. Section 1 points out the different formal verification approaches, their limitations and the approach proposed in this paper. Section 2 presents the case study used in this paper in order to illustrate our approach. Section 3 starts with a presentation of the Caduceus tool and then describes in detail our validation method.

1 Formal Verification

1.1 Model Verification Approach.

A classical approach of formal verification consists in building a model of the system in a formal framework, for instance a theorem prover language, and target properties are proved to be satisfied by this model. This approach can be found for instance in industrial domains, when formal methods are used to increase the security level of products. A model of a given sensitive system is usually built from the system requirements specification. Security policies can then be translated into security properties and proved in the same formal framework.

In this approach, the implementation is generally developed in parallel with the verification process. Therefore the main problem is to justify the link between the verified model and the implementation. This correctness of the model with respect to the source code is mandatory to claim that the code verifies the target

properties. An usual way to strengthen this link is to refine the high level model in lower level models, until a low level model whose link with the code is as straightforward as possible, in terms of data structures and functions. The link between two levels is proved using an abstraction property.

Such a "top-down" refinement approach has been used to prove the correctness of the *Java Card Virtual Machine* embedded into smart cards (see [16]). A high level formal model of the JCVM has been developed in the Formavie Project and security properties such as the confidentiality and the integrity of the embedded applets have been proved on this model (see [3,2]).

A first weakness of the verification at the model level, using refinement, is in terms of optimisation, maintenance, and reusability. Indeed, any modification of the source code needs an update of all the models and formal links. Another weakness is that the last step between the lowest level and the implementation is *informal*. This missing link can be provided by the automatic generation of source code from the formal models, when it is possible. It enables to derive code from the specification after having verified properties on this specification. Such method has been investigated in [8,29,23] with the B tool to generate Java Card or C programs. Also, [14] and [10] proposes an embedded Java Card bytecode verifier, generated from formal models. But this method is not well suited for low level programs, close to the hardware layer, such as operating system programs. These programs are usually written by smart card experienced developers, since some very technical optimisations are usually needed, for instance when managing the memory.

A way to avoid those weaknesses is to consider the source code as the starting point of the verification.

1.2 Source Code Verification Approach.

Several tools for the verification at the source code level exist. One possible approach, taken for instance in the BALI project (see [4] and e.g. [33]), consists in modelling the syntax and semantics of the source code in a proof assistant, using a so-called *deep embedding*, and in proving general theorems on the language. This approach is well suited for meta-theoretical studies but is less practical for actual development of verified code by developers.

An alternative approach consists in inserting annotations into source code and in using a proof tool to verify, automatically or interactively, that the code implements the properties defined by the annotations. Annotations are usually special comments inserted in the source file which can be ignored by the compiler but recognised by the verification tool. They may usually express preconditions and postconditions of functions, variables modified by functions, loop invariants, global invariants, etc. Annotations may be defined by the programmer, or generated, entirely or partially, from the code. For instance, properties specific to the language, such as out-of-bounds array access, can be statically deduced from the code.

Following this idea, several tools have been developed, in particular for programs written in Java. The Java Modeling Language JML ([25]) is a formal

annotation language, that can be analysed by different tools in order to produce documentation, perform dynamic tests and handle properties verification. It is used for the verification of Java programs in the tools ESC/Java ([17,18]), LOOP ([26]), Jack ([12,11]) or Krakatoa ([27]). On the other hand, the Key tool ([1]) proposes an UML based specification for the verification of Java Card applications, while the Jass tool ([5]) is a Design by Contract extension for Java, enabling run-time checks of specification violation, with a possible specification of global properties using traces.

We are interested in a similar approach for C programs. A lot of tools allow to do static analysis of C code (see [31]) but few of them handle explicit preconditions and postconditions. However, the Caveat tool ([15]) provides semi-automatic verification of C programs, where the annotations are built separately from the code. In this paper, we shall use the Caduceus tool which is a direct adaptation of the Java/JML technology for C programs.

All these tools offer the guarantee that given properties are verified *at the source code level*. But the fact that these properties have to be expressed in the annotation language gives rise to several limitations:

- the annotation language is a first-order predicate logic. Therefore the definition of some properties, such as reachability in data structures, becomes heavy whereas it would be immediate in a higher order language. However, some tools allow to use predicates in the annotations that may be instantiated only in the higher order theorem prover used;
- if the proof of several properties is needed, each function annotation will contain the conjunction of all these properties. Thus the code is more "polluted" and the verification process can be heavier;
- properties expressed using annotations are *local* to the function considered. This is well suited for the verification of *functional* properties, such as "the result of a function must be null". But it is often necessary to prove *global* properties over combination of several functions or high level temporal properties, such as the absence of dead-lock. However, existing methods propose a way to express such global properties in a local way, either within the annotation language, using some variables to represent a global state of the program (see [6,24]), or by introducing new annotated code to be proved, representing the global properties.

1.3 Our Approach.

This paper presents a combination of the previous approaches. We use the annotations in order to define a model of the system and we *prove* that the given implementation of this system corresponds to its model. Then the expected security properties can be checked directly on the verified model, which provides a certain level of abstraction with respect to the code.

This method is used to model and verify an operating system module by annotating each function by the description of its behaviour. The case study is described in the following section.

2 Case Study

2.1 Context

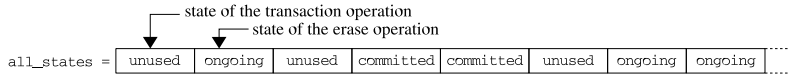
Smart cards are devices where the confidence in the embedded software is crucial. Besides, a smart card needs to be inserted into a reader to obtain power. So if the card is suddenly removed from the reader, the program that was running on card is interrupted. Such a *tearing*, or *power off*, must give rise to coherence verification, stability checks, recovery properties proof, etc. For all these reasons, formal verification is becoming an essential step.

A tearing may have no consequence for some operations. But other operations must be processed *atomically*, i.e., either all instructions of the operation are executed, or none are. This is the case of a *transaction*: if a tearing occurs during the processing of a transaction, all the operations done from the beginning of the transaction must be aborted. Other operations, such as the erasing of a memory segment, need to be *complete* in the sense that they must be resumed or processed again if a power off occurred.

In order to ensure this kind of properties of the "tearing sensitive" operations, variables are usually used to store the current state of the operations. A variable indicates either that the operation has started and is currently *ongoing*, or that it has been *committed*. The variables may also be *unused* if no such operation has yet occurred. In order to model this, we could introduce a set of possible states $state = \{ongoing \mid committed \mid unused\}$ and different variables, such as *transaction_ state* or *erase_ state*, keeping track of the status of the corresponding operations. Then when the card is reset, all states are checked and if some are ongoing, specific measures are taken.

2.2 The Source Code

In our case study, an array `all_states` is used to store the states of all the "tearing sensitive" operations:



For capacity and optimisation reasons, only the smallest space needed to store this information is used. For instance, two bits are sufficient to represent a state:

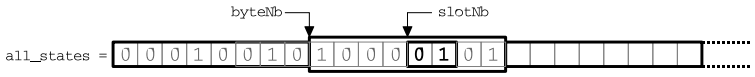
```
#define STATE_UNUSED      0   /* 00b */
#define STATE_ONGOING     1   /* 01b */
#define STATE_COMMITTED   2   /* 10b */
```

Therefore, an unsigned char which contains eight bits may represent four states. The array `all_states` can thus be defined as follows:

```
unsigned char all_states[DIM];
```

where `4*DIM` is large enough to contain states of all tearing sensitive operations.

This optimisation implies that to access a given state in the array, a *byte number* and a *slot number* must be given:



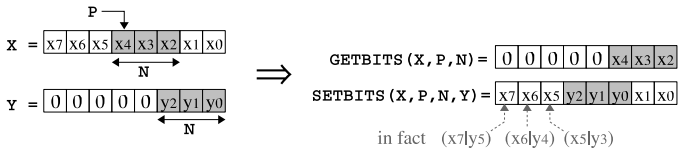
The access functions are defined as follows:

```
unsigned char getState(int byteNb, int slotNb)
    { return GETBITS(all_states[byteNb], 2*slotNb, 2);}
void setState(int byteNb, int slotNb, unsigned char newst)
    { all_states[byteNb] =
           SETBITS(all_states[byteNb], 2*slotNb, 2, newst);}
```

The macros `GETBITS` and `SETBITS` used are defined with bit operations:

```
#define GETBITS(X,P,N)   (X>>(8-N-P)&~(~0<<N))
#define SETBITS(X,P,N,Y)
              ((X|(Y<<(8-N-P)))&(~((~Y&~(~0<<N))<<(8-N-P))))
```

Actually, the macro `GETBITS(X,P,N)` gives the `N` bits from position `P` in the byte `X`. The macro `SETBITS(X,P,N,Y)` returns the byte `X` with the `N` bits that begin at position `P`, set to the rightmost `N` bits of `Y`, leaving the other bits unchanged. More precisely, the leftmost bits of `X` are unchanged *if* `Y` has at most `N` significant bits, i.e. if the integer `Y` is less than 2^N.



2.3 Verification using Existing Approaches

Let us illustrate the approaches presented in Section 1, and more specially their limitations, on the case study presented in the previous section.

Model verification approach. Let us show here that the missing formal link between the model and the code allows to verify properties on the model that are not verified by the code. Usually, for easiness reason, a high level model is used, since it allows to verify high level properties without taking into account low level aspects such as memory allocation. But making this choice increases the risk of an incorrect abstraction. For instance, in our example, an intuitive way to build a model is to define `all_states` as an array of states, where a `state` is an union set of three values: `ongoing`, `committed` and `unused`. The low level aspects of bit manipulation used to retrieve or modify some bits of a byte are abstracted. Therefore some source code bugs may not be detected by any verification on the model. For instance, in the source code, the `SETBITS` macro has the following comment :

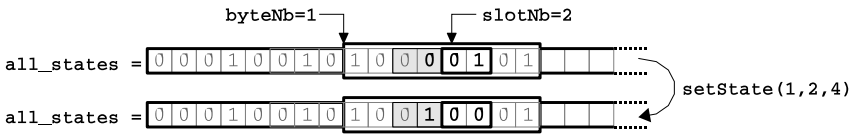
```
/* return X with the N bits that begin at position P set to
   the rightmost N bits of Y, leaving the other bits unchanged */
```

This comment is not correct since it does not mention the condition that `Y` must be less than 2^N. If this condition is not satisfied, the leftmost bits of `X` are

modified. In particular, if the function `setState` is called with a `newst` greater or equal than 2^2, the slots adjacent to the `slotNb` are also modified. In our operating system module, the function `setState` is actually called only with one of the three defined states that are less than 2^2. But if the program is reused, one could define:

```
#define STATE_ABORT_STARTED  3   /* 11b */
#define STATE_ABORT_DONE     4   /* 100b */
```

and use `setState(byteNb,slotNb,STATE_ABORT_DONE)`. This would overwrite the adjacent slots in `all_states`:



This undesirable behaviour would not be detected with a verification on the high level model since the bit operations are not represented after the abstraction:

Actually the model is even incorrect since the adjacent state in the array remains `unused` in the model whereas it becomes `ongoing` in the source code.

Source Code Verification Approach. As already mentioned, the difficulty in handling the verification using inserted annotations consists in the definition of *global properties*. To illustrate this in our case study, here is an example of global property that may not be easily proved using only annotations:

> "*if* `getState(byteNb,slotNb)` *is called just after a call to* `setState(byteNb,slotNb,newst)`, *then the result is* `newst`"

Another example of temporal property is the following: let `commit_next_ongoing` be a function which sets to `committed` the first occurrence of `ongoing` in the array `all_states`. This function would be called at reset and we would like to prove properties such as: "*for any initial configuration of* `all_states`, *there exists a finite sequence of calls to* `commit_next_ongoing` *ending with* `all_states` *containing no* `ongoing` *state*".

3 Source Code Verification of Global Properties

As already mentioned, the idea presented in this paper is to combine the two approaches presented in Section 1 in order to prove that the *source code* verifies some *high level* and *global* security properties. Our method is based on four steps:

1. a specification step: the program is annotated by the specification of its functions. This specification becomes the *local* model of the program.
2. a validation step: the soundness of the local model with respect to the source code is proved using a source code verification tool.

3. a high level modelling step: a memory state transition model, or *global* model, is formally generated from the local model of the code.
4. a security verification step: high level and global security properties are defined and proved to be satisfied by the global model.

We use the Caduceus tool since it offers an architecture which enables such a verification method and meets the requirements needed for our verification of embedded operating system. Indeed it handles C programs and generates explicitly the local model. Caduceus is built on top of the back-end verification tool Why. However, in this paper, we will not do the distinction between the two levels of analysis and refer only to *Caduceus* for actually the combination of both systems. In the following, we present the main aspects of Caduceus (for more details see [21,22,19,20]) and then detail each step of our approach, illustrated on the case study.

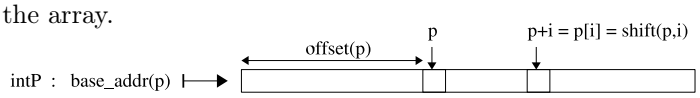
3.1 The Caduceus Tool

Annotations. Caduceus is a verification tool at the source code level. It is based on Hoare logic with preconditions and postconditions, but with an additional explicit interpretation of both the specification and the code as state functions. The programs handled are ANSI C source code, annotated with a specification language inspired by the Java Modelling Language (JML, see [25]). Annotations are used to define functional properties of each function. Formulae are expressed in a first-order language where C expressions without side-effects can be used as well as predicate variables (to be interpreted later) and specific keywords. They may express functions preconditions (with the keyword `requires`), side-effects (with `assigns`), postconditions (with `ensures`), global invariants, loop invariants, loop variants and loop side-effects, logical functions (`logic`) or predicate (`predicate`), etc. Moreover, in the postcondition, the construction `\result` may be used for the result returned by the function and `\old` for the initial state of the function. Finally, the keyword `\nothing` can be found in the assign clause to state that the function has no side-effect.

Translation. Caduceus interprets a C program using a memory model. Instead of modelling the memory as a big array, Caduceus follows Burstall and Bornat's approach (see [9,13]) where a spatial separation divides the memory into disjoint memory locations whenever it is possible: for instance, two different fields of a structure will be in separated memory locations. This separation ensures "for free" that changes made in one memory location do not affect the other locations. Within a single memory location, the separation of variables is also ensured, in the sense that a *proof obligation* is generated whenever the separation is not clearly established.

The model identifies the notions of pointer and array. Hence, the *basic values* are either direct values in numeric types (integers or reals) or pointers. A value `p` of type `pointer` is either the `null` pointer or a pair `(base_addr(p),offset(p))` made of the address of the memory block containing `p` and the offset of `p` within

this block. Then, the memory state of the C program is represented by a set of global variables corresponding to statically separated memory spaces. Each memory space maps pointers to values. For instance, a variable `intP` may be used to represent the part of memory where arrays of integers are allocated. We can visualise `intP` as a function which associates each base address corresponding to an allocated array of size n to a piece of memory of size n containing the integer values in the array.



Caduceus provides an access function `acc(intP,p)` retrieving the value pointed by `p` in the state `intP` and a modification function `upd(intP,p,3)` whose result is a new memory state `intP'` where the value pointed by `p` becomes 3. Moreover, as shown in the figure, a pointer arithmetic function `shift(p,i)` allows to represent the C expression `p+i` or `p[i]`. Finally an additional variable `alloc` represents an *allocation store* which tells which addresses are allocated and the size of the block it points to. This allows annotations such as `(valid alloc p)`, or `(valid_range p i j)`, etc.

Another aspect that must be taken into account by Caduceus is the *effects inference*. Caduceus computes for each C function the set of memory variables and global variables which are read and/or modified.

To conclude, Caduceus interprets each C construction as a functional transformation of values of memory states, using a monadic interpretation.

Verification. The Caduceus tool generates *verification conditions*. These are the missing parts of the verification process that must be proved in order to ensure the soundness of the program with respect to the specification given in the annotations. A specific aspect of Caduceus is its independence with respect to the prover. Therefore the verification conditions may be checked in any of the theorem provers proposed (PVS [28], Simplify [30], Coq [32], ...)

Due to its interactive and higher order aspects, the Coq theorem prover (see [32]) has been chosen for our verification method. Moreover, when used with Coq, Caduceus provides a *validation term* ensuring the correctness of each function in the program, which is useful for our method, as explained Section 3.4. The validation term is a proof of $\forall \mathbf{x}.\ Pre(\mathbf{x}) \rightarrow \exists \mathbf{x'}.\ Post(\mathbf{x}, \mathbf{x'})$ where $\mathbf{x}$ and $\mathbf{x'}$ represents the values of memory variables modified by the function before and after the function call. Assuming the input memories $\mathbf{x}$ satisfies the precondition, an output state can be reached which satisfies the postcondition. In the Coq system, which is based on Type Theory, the validation term corresponds to an executable functional term which represents our semantics of the given C program. Type checking this validation term therefore ensures the correctness of the program with respect to its specification[3]. See [20] for a more detailed analysis of this technology.

[3] This is of course under the condition of the correctness of Caduceus functional interpretation of C programs.

3.2 Specification Step

The annotations are used in order to describe the function behaviour, i.e., its specification. In our case study, the postcondition of `getState` must indicate that the result of the function contains the two bits at position `2*slotNb` in the byte `all_states[byteNb]`. Talking about a given bit of an integer in the annotation language is quite impossible. Moreover, a Coq library provides support for binary representation of integers. Therefore, we define a logical function `GetBits`, which is declared in the annotations, and will be instantiated in the Coq language:

```
/*@ logic int GetBits(int b,int p,int n) */
```

This function represents the same operation as the `GETBITS` macro, i.e., it gives the `n` bits from position `p` in the byte `b`. Once declared, this function can be used in any annotation. On the other hand, the precondition of `getState` indicates that the indexes `byteNb` and `slotNb` are valid in the array `all_states`. Finally, `getState` does not modify any global variable. The specification of `getState` can thus be expressed as follows:

```
/*@ requires (0<=byteNb<DIM) && (0<=slotNb<8/2)
  @ assigns \nothing
  @ ensures \result == GetBits(all_states[byteNb],2*slotNb,2)*/
unsigned char getState(int byteNb, int slotNb)
   { return GETBITS(all_states[byteNb], 2*slotNb, 2);}
```

For the `setState` function, an additional precondition must be added mentioning that the new state must be less than 2^2. Concerning what is modified by `setState`, it consists of two bits of `all_states[byteNb]`. However, the assign clause may not represent the bits of an integer. Therefore the assigns clause will contain the whole byte `all_states[byteNb]` and the postcondition is used to indicate that the two bits at position `2*slotNb` become equal to the `newst` given and that the other bits are unchanged:

```
/*@ requires (0<=byteNb<DIM) && (0<=slotNb<8/2)
  @       && (0<=newst<2^2)
  @ assigns all_states[byteNb]
  @ ensures (GetBits(all_states[byteNb],2*slotNb,2)==newst)
  @     &&  ( \forall int j;  0<=j<8/2 && (j!=slotNb) =>
  @             GetBits(all_states[byteNb],2*j,2) ==
  @             GetBits(\old(all_states[byteNb]),2*j,2) )  */
void setState(int byteNb, int slotNb, unsigned char newst)
   { all_states[byteNb] =
         SETBITS(all_states[byteNb], 2*slotNb, 2, newst);}
```

3.3 Validation Step

The validation step consists in proving the verification conditions generated by Caduceus. When used with Coq, the verification conditions are lemma statements that may be proved interactively. In our example, the main goals to be established concern the postconditions of the two functions. For the `getState` function, the following goal has to be proved:

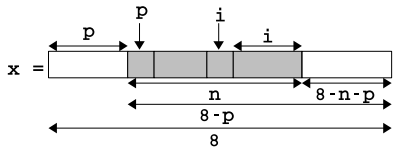
```
\result == GetBits(all_states[byteNb],2*slotNb,2)
```

where `\result` is built by the macro `GETBITS`, i.e., it consists in a combination of binary operations. Concerning `setState`, two goals have to be proved:

```
1. (GetBits(all_states[byteNb],2*slotNb,2)  ==  newst)
2  \forall int j;  0<=j<8/2 && (j!=slotNb) =>
      GetBits(all_states[byteNb],2*j,2) ==
      GetBits(\old(all_states[byteNb]),2*j,2) )
```

All these goals are equality statements between two bytes. The idea is to do the analysis at a bit level, using Coq libraries defining binary representation of integers and binary operations. In other words the goals are proved using an auxiliary lemma stating that two bytes are equal if all their bits are pairwise equal. This needs the definition of a function computing the *ith* bit of a given byte. Moreover, lemmas are needed to compute the *ith* bit for all binary operations, in order to obtain the *ith* bits of `\result`.

Finally, the function `GetBits` that has been only declared is defined using the Coq library. Then we need to know the *ith* bits of the byte resulting from the `GetBits` operation: `(ith (GetBits x p n) i) = (ith x (i+(8-n-p)))` for any `x`, `p`, `n` and `i` such that ($0 \leq$`p`<8), ($0 \leq$`n`$<$`8-p`), and ($0 \leq$`i`$<$`n`).



3.4 High Level Modelling Step

Our goal is to be able to express high level global properties. Such properties are defined in terms of states at function call and states resulting from the function execution. Examples of higher order properties are:

- *"if the state before the call to the function f satisfies P, then there exists a state resulting from a finite sequence of calls to f which satisfies Q"*
- *"if the state s before the call to the function f satisfies P and s' is the state resulting from the execution of f from s, then the call to the function g from the state s' results in a state satisfying Q".*

Therefore we would like to model a function as a *transition* relation between two memory states. In other words, we would like to define, for each given function f, a binary relation $f\_transition$ such that $\mathbf{x}$ is in relation with $\mathbf{x}$' by this relation (denoted by ($f\_transition$ $\mathbf{x}$ $\mathbf{x}'$)) if $\mathbf{x}$' is the state resulting from the execution of f from the state $\mathbf{x}$.

The identification of the memory states $\mathbf{x}$ and $\mathbf{x}$' depends on the memory model. The memory model chosen here is the one defined in Caduceus, described in Section 3.1. A memory state is made of the global variables of the program (numeric values or references), the global variables corresponding to the memory segments (e.g. `intP`) and a variable *alloc* storing the allocated addresses.

More precisely, for each function f with a list $\overrightarrow{a}$ of parameters, Caduceus computes the set $\overrightarrow{z}$ of "read-only" variables (variables of the program and variables representing memory segments) and the set $\overrightarrow{t}$ of "read-written" variables. Caduceus also computes the precondition $Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t})$ from the *requires* clause of the annotation and the memory states computed. In the same way, the postcondition $Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})$ is computed, where $\overrightarrow{t}$ corresponds to the values after the function call and $\overrightarrow{t@}$ to the values at function call. The postcondition also includes the *assign* clause of the function.

For instance, for the function `setState`, a single state variable $intP$ is introduced, corresponding to the memory segment where all_states is allocated. The list of parameters $\overrightarrow{a}$ is $(byteNb, slotNb, newst)$, the list of read-only variables $\overrightarrow{z}$ is $(all\_states, alloc)$ and the list of read-written variables $\overrightarrow{t}$ is $(intP)$. The precondition contains the *requires* clause, together with a validity condition of the variable all_states in the allocation table $alloc$:

$$(0 \leq byteNb < DIM)\ and\ (0 \leq slotNb < 8/2)\ and\ (0 \leq newst < 2^2)\ and\ (valid\_states\ all\_states\ alloc)$$

Finally the postcondition combines the *assigns* and the *ensures* clauses:

$$\begin{array}{l}(GetBits(acc(intP, shift(all\_states, byteNb)), 2 * slotNb, 2) = newst)\ and\\ (\ \forall\ j : int.\ (0 \leq j < 8/2\ \ and\ \ j \neq slotNb)\ \rightarrow\\ \quad (GetBits(acc(intP, shift(all\_states, byteNb)), 2 * j, 2) =\\ \quad\ GetBits(acc(intP@, shift(all\_states, byteNb)), 2 * j, 2))\ )\ \ and\\ assigns(alloc, intP@, intP, pointer\_loc(shift(all\_states, byteNb)))\end{array}$$

Using the memory states computed by Caduceus, there are two approaches for the *transitional definition* of a function f: we may use the *code* of the function or only its *specification*. Using the code means defining the resulting state **x**' as the translation of the *code*: **x**'$=\bar{f}$(**x**), where $\bar{f}$ represents the functional translation of the C program. This approach may be useful if some computational aspects of the function are needed to prove a specific property and are not represented in the specification of the function. However this needs an explicit functional interpretation $\bar{f}$ of the code, which may be huge, giving rise to heavy proofs. Let us note that, in addition, this model of the code is provided by Caduceus only when used with the Coq prover (see Remark below).

We choose a more abstract approach following the idea that since we proved that the specification represents the program, the function can be modelled only by its specification. In other words, **x** and **x'** are in relation by $f\_transition$ if **x** verifies the precondition of f and **x'** verifies its postcondition:

$$(f\_transition\ \mathbf{x}\ \mathbf{x'}) \quad \equiv \quad Pre_f(\mathbf{x})\ \wedge\ Post_f(\mathbf{x}, \mathbf{x'})$$

More precisely, using the work performed by Caduceus in order to identify **x** and **x'**, $f\_transition$ has the following form:

$$(f\_transition\ result\ \overrightarrow{a}\ \overrightarrow{z}\ \overrightarrow{t@}\ \overrightarrow{t}\ ) \equiv Pre_f(\overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}) \wedge Post_f(result, \overrightarrow{a}, \overrightarrow{z}, \overrightarrow{t@}, \overrightarrow{t})$$

Of particular note is the prover independence of the method itself, since annotations are first order formulae. However, targeted properties will have to be expressed in the chosen prover. Therefore, in the case of complex temporal prop-

erties, such as properties on transitive closures, higher order provers will be more suited.

Going further into technical aspects, Caduceus does not actually give access to the functional translation of the precondition and the postcondition directly, but only to the *validation term*, and *only* when used with the Coq theorem prover. The validation term has the following type:

$$f\_valid \ : \ \forall \mathbf{x}.\ Pre_f(\mathbf{x}) \ \rightarrow \ \exists \mathbf{x}'.\ Post_f(\mathbf{x}, \mathbf{x}')$$

However, a trick of the Coq language using type inference allows to express the property $Pre_f(\mathbf{x}) \ \wedge \ Post_f(\mathbf{x}, \mathbf{x'})$ as a simple expression only using f_valid.

Remark: in the first approach mentioned, using the functional interpretation of the *code*, the memory state **x'** is actually the witness built by the validation term. Therefore, this approach is also possible only when Caduceus is used with the Coq prover.

The fact that the validation term is only provided with the Coq prover is a major argument in our choice of Coq. Another argument being an easier definition of global or temporal properties.

3.5 Security Verification Step

In this final section, we show that the global properties mentioned in Section 2.3 can be expressed and proved using our global model.

The first property was: "*if* `getState(byteNb,slotNb)` *is called just after a call to* `setState(byteNb,slotNb,newst)`, *then the result is* `newst`". This property has the following statement in Coq:

```
Lemma get_set :
   forall (byteNb slotNb newst:Z)(all_states:pointer)
          (alloc:alloc_table)(intP:memory Z)(intP0:memory Z),
   (setState_transition byteNb slotNb newst all_states alloc
                        intP intP0)
-> (getState_transition byteNb slotNb all_states alloc
                        intP0 newst).
```

The proof is straightforward after unfolding the transition definition.

The second property was: " *for any initial configuration of* `all_states`, *there exists a finite sequence of calls to* `commit_next_ongoing` *ending with* `all_states` *containing no* `ongoing` *state*". The specification of `commit_first_ongoing` states that either there was already no `ongoing` state in `all_states` and then nothing is done, or the first `ongoing` state of `all_states` before the call is changed into a `committed` state. In Coq, a finite sequence of calls to such a function can be defined inductively using the transitional formal model of the function. Then the Coq statement of the property states that for any array `all_states` and initial state `intP`, there exists a memory state `intP0` such that this state results from the successive calls to `commit_first_ongoing` from `intP` and that there is no `ongoing` state in `all_states` in the state `intP0`. The memory state `intP0` given in

the proof is the witness of the validation term and the proof is done inductively on the size of the sequence of calls: one step of the function makes the number of `ongoing` states decrease by one, therefore, after a finite number of calls, this number reaches zero.

We presented here simple global properties since the case study has been shortened for the illustration. Thus only few functions were presented. But once the whole system is specified and its model is generated, other global security properties concerning the behaviour of the entire system may be proved.

4 Conclusion

In the smart cards world where *security* and *performance* are the main business criteria, formal verification activity becomes a mandatory step. Building high level models of the system being developed to prove correctness properties is useful but is still expensive, as it requires experts. Moreover, a formal link between the models and the actual system implementation is lacking. The goal is then to build tools generating secure code from verified high level models. But those tools have to be improved to take into account the scarce resources of smart cards. Another immediate solution is to reason directly on the source code. This method could be handled by the developer, but reasoning at this low level limits the expressiveness of the properties to prove.

The method we proposed here allows to combine the two approaches and to take benefits from both. A functional verification is performed at the source code level by the insertion of annotations describing the expected behaviour of the program. This step strengthen the confidence in the code by providing a proof that its execution will have the expected behaviour. The originality of our method is to use the program specification already defined in the annotations, to derive a high level model allowing the definition and verification of high level security properties. The model is thus *automatically* generated from an *existing* formal specification. Moreover, the missing formal link between the model and the code is provided by the formal derivation of the model from a formal specification, together with the formal proof that this specification is verified by the code. Therefore global security properties concerning the behaviour of the whole system can be proved on the model, in a independent way.

Our future work will consist in generalising the method in order to handle a wider class of embedded programs and to be able to express a wider range of smart card security properties. For instance, casts of pointer or structure are used in our embedded source code, but this is the main unsupported feature of Caduceus. This is due to the memory separation model used in Caduceus, that becomes incorrect in the presence of such casts. The memory model must therefore be adapted to handle any embedded source code. Another extension would be to represent the tearing in the annotation language. This would allow to define the conditions that must hold even if a tearing occurs. Since the high level model is derived from the annotations, global properties could then be proved concerning the global behaviour of the system in the case of a power off. An

interesting direction would also be to investigate an automatic transformation of temporal security properties into properties expressed on our high level model. In this context, a comparison with model checking based methods, which is missing in this paper, should be made.

Finally, our method allows a faster transfer of the tools to the developers, giving them the possibility to define properties directly on their source code. This will help us to achieve our main goal of a wide deployment of formal verification tools to the developers to produce automatically a secure embedded code.

Acknowledgements. We would like to thank Jean Christophe Filliâtre, Thierry Hubert and Claude Marché for their useful help and support in using Caduceus.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print. `http://www.key-project.org/`.
2. J. Andronick, B. Chetali, and O. Ly. Formal Verification of the Integrity Property in Java Card Technology. In *International Conference on Research in Smart Cards (Esmart'03)*, September 2003.
3. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *LNCS*, pages 335–351. Springer-Verlag, September 2003.
4. The Bali project. `http://isabelle.in.tum.de/bali/`.
5. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, July 2001. `http://csd.informatik.uni-oldenburg.de/~jass/`.
6. G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and M. Pavlova. Enforcing High-Level Security Properties For Applets. In *Sixth Smart Card Research and Advanced Application IFIP Conference (CARDIS'04)*, August 2004.
7. G. Barthe, G. Dufay, L. Jakubiec, B. P. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP'01)*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.
8. D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 94–113. Springer-Verlag Heidelberg, October 2005.
9. R. Bornat. Proving Pointer Programs in Hoare Logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC'00)*, pages 102–126. Springer-Verlag, 2000.
10. L. Burdy, L. Casset, and A. Requet. Formal Development of an Embedded Verifier for Java Card Byte Code. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, pages 51–58. IEEE Computer Society, 2002.

11. L. Burdy, J.-L. Lanet, and A. Requet. Java Applet Correctness: A Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, September 2003.
12. L. Burdy and A. Requet. Jack : Java Applet Correctness Kit, November 2002.
13. R. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence*, 7:23–50, 1972.
14. L. Casset. Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods. In L.-H. Eriksson and P. Lindsay, editor, *Proceedings of the International Symposium of Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 290–309. Springer-Verlag, 2002.
15. The Caveat Project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html/`.
16. B. Chetali, C. Loiseaux, E. Gimenez, and O. Ly. An Interpretation of the Common Criteria EAL7 level : Formal Modeling of the Java Card Virtual Machine. In *3rd International Common Criteria Conference (ICCC'02)*, May 2002.
17. ESC/Java. `http://research.compaq.com/SRC/esc/`.
18. ESC/Java2. `http://www.sos.cs.ru.nl/research/escjava`.
19. J.-C. Filliâtre. The Why Verification Tool. `http://why.lri.fr/`.
20. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
21. J.-C. Filliâtre and C. Marché. The Caduceus tool for the Verification of C Programs. `http://why.lri.fr/caduceus/`.
22. J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29, Seattle, November 2004. Springer-Verlag.
23. A. Hammad, A. Requet, B. Tatibouët, and J.-C. Voisinet. Java Card Code Generation from B Specifications. In J. S. Dong and J. Woodcock, editors, *5th International Conference on Formal Engineering Methods (ICFEM'03)*, volume 2885 of *LNCS*, pages 306–318. Springer-Verlag Heidelberg, November 2003.
24. M. Huisman and K. Trentelman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.
25. G. T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: Notations and Tools Supporting Detailed Design in Java. In *OOPSLA 2000 Companion*, pages 105–106. ACM, October 2000.
26. Loop. `http://www.sos.cs.ru.nl/research/loop`.
27. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa Tool for Java Program Verification, 2002. `http://krakatoa.lri.fr/`.
28. The PVS system. `http://pvs.csl.sri.com/`.
29. A. Requet and G. Bossu. Embedding Formally Proved Code in a Smart Card: Converting B to C. In *Third International Conference on Formal Engineering Methods (ICFEM'00)*, pages 15–24. IEEE Press, 2000.
30. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/Simplify.1.html`.
31. Static Source Code Analysis Tools for C. `http://www.spinroot.com/static/`.
32. The Coq Development Team LogiCal Project. *The Coq Proof Assistant Reference Manual*. `http://pauillac.inria.fr/coq/doc/main.html`.
33. D. von Oheimb and T. Nipkow. Machine-checking the Java Specification: Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer-Verlag, 1999.